Using Games to Understand and Create Randomness

Henno, Jaak; Jaakkola, Hannu; Mäkelä, Jukka Heikki Antero

Published in:
SQAMIA 2018 - Proceedings of the 7th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications

Published: 01.01.2018

Document Version
Publisher's PDF, also known as Version of record

Citation for pulished version (APA):
Henno, J., Jaakkola, H., & Mäkelä, J. H. A. (2018). Using Games to Understand and Create Randomness. In Z. Budimac (Ed.), *SQAMIA 2018 - Proceedings of the 7th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications: Proceedings of the 7th Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications* (Vol. 2217, pp. 1-9). CEUR Workshop Proceedings. CEUR workshop proceedings Vol. 2217

Using Games to Understand and Create Randomness

JAAK HENNO Tallinn University of Technology
HANNU JAAKKOLA, Tampere University of Technology
JUKKA MÄKELÄ, University of Lapland

Massive growth of data and communication encryption has created growing need for non-predictable, random data, needed for encryption keys creation. Need for randomness grows (nearly) linearly with growth of encryption, but randomness is very important ingredient also e.g. in quickly growing industry of game programming. Computers are deterministic devices and cannot create random results, computer procedures can generate only pseudo-random (looking random) data. For true randomness is needed some outside information – time and placement of user's keystrokes, fluctuations of current, interrupt requests in computer processor etc. But even those sources can often not comply with requests from our increasingly randomness-hunger environment of ciphered communications and data.

Growing need for randomness has created a market of randomness sources; new sources are proposed constantly. These sources differ in their properties (ease of access, size of required software etc.) and in ease of estimating their quality.

However, there is an easily available good source for comparing quality of randomness and also creating new randomness – computer games. The growing affectionateness of users to play digital games makes this activity very attractive for comparing quality of randomness sources and using as a source of new randomness. In the following are analyzed possibilities for investigating and extracting randomness from digital gameplay and demonstrated some experiments with simple stateless games which allow to compare existing sources of (pseudo) randomness and generate new randomness, which can be used e.g. to create cyphering keys in mobile and Internet of Things devices.

1 INTRODUCTION

All businesses and private persons are increasingly reliant on many formats of digital data, which is essential in our personal lives, economic prosperity and security. But criminals have also understood that digital data has become the most valuable resource. Data breaches are increasing by more than 20 percent in a year [BSA. Encryption 2018] and they have become the most worrying feature of Internet [Fortinet 2018].

The main method to protect our data and communications is encryption. Use of encryption for data and communications protection is growing rapidly. Cisco, the largest networking company in the world shows that encrypted traffic has increased by more than 90 percent year over year [Cisco 2018]. Currently already at least half of websites are encrypting traffic and research and advisory company Gartner predicts that by 2019 already 80 percent of web traffic will be encrypted [Gartner 2018]. The e-mail encryption market is expected to have 24% annual growth through 2020 [SendItCertified 2018]. With advances of cloud computing have many enterprises started themselves to create and manage their encryption keys [Ponemon 2018] - the Bring Your Own Key (BYOK) practice allows enterprises to retain control of their encryption keys.

Every encrypted bit in a message is somehow transformed by (at least one) bit of encryption code. For different messages different encryption bits should be used – if some encryption scheme is repeated often it becomes easier to break. Thus the need for randomness grows (nearly) linearly with the amount of encrypted traffic and data.

New computing environments - the coming era of IoT (Internet of Things), virtual/cloud servers etc. all increase need for randomness, thus there are already proposals for special services even from governments [Chen2018] to serve entropy, i.e. random data [NIST 2016]. In order to deliver provided entropy to users is proposed a special new protocol, '*Entropy as a Service Protocol*' [EaaSP 2018]. But

Author's address: J. Henno, Tallinn University of Technology, Ehitajate tee 5, 19086 Tallinn, Estonia; email: jaak.henno@ttu.ee; H. Jaakkola, Tampere University of Technology, Pervasive Computing, P.O. Box 300, 28101 Pori, Finland; J. Mäkelä, University of Lapland, P.O. Box 122, 96101 Rovaniemi, Finland, email: jukka.makela@ulapland.fi.

In: Z. Budimac (ed.): Proceedings of the SQAMIA 2018: 7th Workshop of Software Quality, Analysis, Monitoring, Improvement, and Applications, Novi Sad, Serbia, 27–30.8.2018. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

for delivery the entropy should also be encrypted, thus this is a new source needing 'fresh' entropy. Therefore it is not clear, whether this service will reduce the need for entropy or contrary, increase it. Everything is much simpler if the entropy is generated and collected there where it is needed.

Here is proposed a use of games for creating entropy/randomness and for comparing quality of randomness (unpredictability) of already established sources. The inputs for the generated sequence are either human's gameplay or an already established source of randomness, the output – computer-generated sequence of moves or the sequence of game states, i.e. pairs (arrays in multiplayer mode) of inputs from all players. The main idea is that storing sequence of moves made during the game and using the maximally similar (to the current one) situation from this history allows computer to improve its play, produce better responses (moves) and also produce better randomness. Adding the (unpredictable) component of human gameplay improves randomness of the output, which can be further improved with iterating the process.

In the next section 2 is given overview of currently used methods to create randomness in computers and problems with establishing, whether a sequence of numbers is random (unpredictable). In section 3 is described a simple game, which requires producing uniformly distributed random binary sequences. In section 4 is presented a principle of learning from the maximally similar situation in gameplay history and in section 5 described acting on this principle computer algorithm, which makes computer player superior against human players. The algorithm can be used also against established sources of randomness, i.e. to evaluate their quality and its output can also be used as a new source of randomness. In the section 6 are described results from experiments and tests. In the section 7 is presented a generalization of the previous results employing binary sequences to m-ary sequences, m > 2.

2 RANDOMNESS AND INFORMATION

It is impossible to generate random values using computers basic operations – computer is a deterministic device and all programs return determined value (if not, then the computer is severely broken). All computers are finite devices and after a sufficiently long time they enter a loop, will repeat already generated values, which definitely are not random. John von Neumann commented on this: "Anyone who attempts to generate random numbers by deterministic means is, of course, living in a state of sin." Algorithms used on programming language's translators for generating random values are actually pseudorandom number generators (PRNG) - big loops, which after their period start repeating output values. One of the best known is the formula (a linear congruential generator) used in popular standard C-language library glibc, which produces in a pseudorandom order a cycle with length $2\hat{}31$ using the recurrence:

$$x_{i+1} = (1103515245 * x_i + 12345) \bmod 2^{\circ}31$$

The cycle length of the Mersenne Twister, the default PRNG in many computing environments is even bigger - $2^{19937} - 1$. But the size of these loops does not guarantee that generated numbers are unpredictable - the more computers use the same PRNG, the less unpredictable it comes. And sometimes randomness is just faked by a finite list of integers, e.g. in the very popular game Doom randomness was introduced by a list of 256 integers [Doom 1997].

A seemingly good source of randomness is time and user inputs. In first computers, which did not have time functions, games got randomness e.g. counting game frames which were already played before user pressed "Start" or performed some other actions. But this kind of sources are very limited, besides, players tend to perform their actions in similar temp, thus after some plays 'random' becomes 'non-random'. Use of the time has also several problems. Many computing processes are started by

timer, i.e. in pre-defined time, thus all derivatives of 'time' also become predictable. Predictable are also many other seemingly random values extracted from computer software and/or hardware. The 'father of all browsers' Netscape used time, the process and the parent process's ID-s for seeding its PRNG, but the resulting values were shown to be relatively predictable. Many programming environments even do not show the most precise value of time available or present it in different formats. In most browsers the JavaScript function

Date.now();

produces integer value, but in Firefox the result is rounded to even value. Another time function

performance.now();

produces in Firefox even integer, but in Chrome and IE 11 – a real number with 12 decimals.

We use and check randomness with finite devices and when their memory (number of states) grows, some previously hidden regularities appear thus 'random' becomes 'non-random'. Many formulae for producing seemingly random values are at their introduction considered 'good enough', but later found to be not 'good enough' as e.g. the RC4 (Rivest Cipher 4) which is/was used in several commonly used encryption protocols and standards, e.g. in the TLS (Transport Layer Security), but was some years ago prohibited; widely known was periodicity in the random function of Microsoft PHP translator.

Randomness is an evasive concept to define. The widely accepted definition is the Kolmogorov-Chaitin definition [Kolmogorov 1965]:

a sequence is random if it can't be expressed by any algorithm or device which can be described using less symbols than what are in the sequence.

This definition and consequent definitions [Martin-Löf 1966], [Schnorr 1971] are theoretical, using infinite sets of concepts ('*any algorithm*'), thus useless for evaluating quality of a source of randomness. Available randomness tests [NIST 2016], [Marsaglia 2002] require installation of specific software and from user higher than average computer skills.

3 GAMES

Randomness is also very important in games - a rapidly growing area of software development. Humans are not very good sources of randomness, but employing different aspects of human gameplay, e.g. human errors allows to create quite good, unpredictable randomness [Alimomeni 2014]. In the following are analyzed possibilities for comparing quality of various sources of randomness and demonstrated experiments with simple static games which allow to generate quite good randomness.

For creating randomness and investigating randomness created during play could be used static games, where the game data structure does not change (not like in chess) and the game simply checks correspondence of player's inputs to rules, defining the result; the rules do not change during the play. Such a game is just a finite table for determining wins, loses and draws.

There are numerous examples of such games, e.g. the game 'even-odd' (or 'Matching pennies' [Matching pennies]): two players (call them 'even' and 'odd') simultaneously select one of two options '0' or '1' (integers); 'even' wins, if the sum of their selections is even, 'odd' – if it is odd.

Table. 1. The decision table for the game 'even-odd'.

'even' / 'odd'	'0'	'1'
'0'	win = 'even'	win = 'odd'
'1'	win = 'odd'	win = 'even'

Game decision tables have often various symmetries, e.g. the above table allows reflection – change of rows to columns and vice versa (players have similar possibilities).

Players select their moves with certain probabilities, collection of these probabilities is called player's strategy. Suppose the strategy of the player 'even' for selecting '0' or '1' is

$$p_e = [p_{e0}, 1 - p_{e0}]$$

Here p_{e0} is the probability that player 'even' selects '0' and $1 - p_{e0} = p_{e1}$ - probability, that he selects '1'. The strategy for player 'odd' is

$$p_o = [p_{o0}, 1 - p_{o0}]$$

Player 'even' wins, if both players select the same values and loses, if the select different values, thus he wants to maximize probability of winning, i.e. the quantity

$$P_{eW} = p_{e0}p_{o0} + (1 - p_{e0})(1 - p_{o0}) = 1 - p_{e0} - p_{o0} + 2p_{e0}p_{o0}$$

The probability of winning for player 'odd' is

$$P_{oW} = p_{e0}(1 - p_{o0}) + (1 - p_{e0})p_{o0} = p_{e0} + p_{o0} - 2p_{e0}p_{o0}$$

The probability of outcome for any of players is

$$W = P_{eW} - P_{oW}$$

The expression for probability of win for player 'even' is a function of its probability p_{e0} to select '0'. A function is growing if its derivative is positive and in the maximum – equals to zero, thus for player 'even' the best strategy would be:

$$\frac{dP_{eW}}{dp_{e0}} = p_{o0} - (1 - p_{o0}) - (1 - p_{o0}) + p_{o0} = 0$$

From here it follows

$$p_{o0} = 1/2$$

Thus $p_{o1} = 1 - p_{o0} = 1/2$ and from the symmetry of the game also $p_{e0} = p_{e1} = 1/2$.

This is the 'Nash Equilibrium' [Nash 1950] – an optimal strategy if other players do not change their strategy. Nash proved existence of equilibrium points, but did not show, how to find them. It turned out to be quite a complex problem [Babichenko Rubinstein 2016], but for this very simple game it was easy to calculate the optimum.

When playing, players generally do not take derivatives to calculate their optimum strategy – they come to the optimum with step-by-step modifications of their strategies. Knowing, that both '0' and '1' should be selected with the same probability ½ does not help much – they should be selected so that opponent could not detect any pattern in selections and it is very easy to 'slip down' from the equilibrium point, since this is a saddle point of the function W :

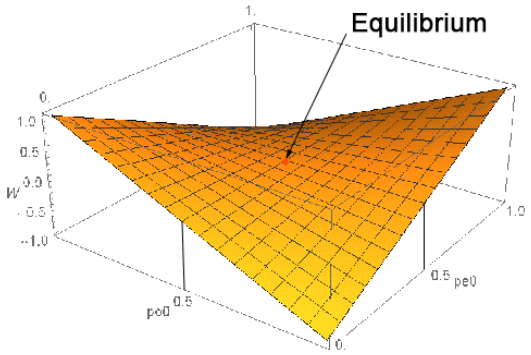

Fig. 1. The 3D plot of the function W and the equilibrium point – change of strategy for any player will change probability of winning for both players.

4 LEARNING

Real (human) players usually quickly understand, that they should follow what the opponent does – they should learn from the sequence of already made moves.

A player with very limited memory considers only the last move and adheres the simplest learning strategy (learning with deph 1):

if I won – repeat the last move; if I lost – change the move.

Consider this game as a finite automaton, where states are pairs $[0,0],[0,1],[1,0],[1,1]$ of players moves (first the move of player 'even', then – player 'odd), transforming the current state to the state designated with this move and producing outputs '*e*','*o*' which indicate, who won (player 'even' or player 'odd'), the output is shown after the state. It is easy to see, that whatever was the first move (where they could yet not follow the above rule), the game converges to four-step cycle. For instance, suppose the first move was [0,0] and both players follow the above rule:

$$[0,0]e \xrightarrow{0,1} [0,1]o \xrightarrow{1,1} [1,1]e \xrightarrow{1,0} [1,0]o \xrightarrow{0,0} [0,0]e \rightarrow \ldots$$

Whatever was the first move, players always create this four-state deterministic automaton. Thus a wise player tries to confuce the opponent, make random moves and learn from previous game situations. Every 'novel' move, a move that has not been used in current situation before adds to the above automaton a new transition and the automaton becomes non-deterministic.

The situation is similar when two finite (deterministic) automata interact and try to quess each other's behaviour ('pwn' the opponent, [Henno 2017]). In order to learn opponents behaviour automata store interaction history (moves). If an automaton has used all its memory it starts looping, repeat its moves (it is deterministic). Thus if other automaton has more memory and discovers that opponent is looping it knows from thereon all actions (moves) of the other.

5 ALGORITHM

The algorithm is based on seeking loops in opponent's behaviour (learning with depth > 1):

- look back and when you see a situation maximally similar to the current one make the move that then (in the previous similar situation) would make you winning.

More precisely: suppose the sequence of moves from the first state S_0 to the the current state S_c of the game is

$$S_0, S_1, \ldots, S_n, \ldots, S_k, S_c, S_{c+1}, \ldots, S_n, \ldots, S_k, S_c$$

From the current state s_c search backwards for the longest subsequence which already has occurred before the current state s_c; suppose it is $s_n,\ldots,s_k,s_c$:

$$s_0,s_1,\ldots\underline{|s_n,\ldots,s_k,s_c}\,s_{c+1},\ldots,\underline{|s_n,\ldots,s_k,s_c}$$

For instance, for the list (for interactions where are used digits)

$$1,7,2,1,2,3,4,2,1,2,\underline{|4,3,1,2,3,5}\,,7,5,9,8,9,1,2,3,4,7,,\underline{|4,3,1,2,3,5}$$

the longest suffix which the algorithm would return is

$$4,3,1,2,3,5$$

Thus the next move should beat the opponents earlier move in this situation: '7'.

The longer the subsequence is, the more probable is that the opponent (with limited memory) will repeat its last move in this situation, i.e. the move that created the state s_{c+1}. Now select your move correspondingly, i.e. if the state s_{c+1} was for you winning, repeat your move; if it was not, change it.

6 CREATING RANDOMNESS

In tests with students (and authors of this paper) this simple strategy for computer has turned out to be very good – if the length of the game is >50, human players are already behind (you can test yourself at http://staff.ttu.ee/~jaak/games/). We can not remeber long sequences of moves and we are not sufficiently random to beat computer, especially if the memory requirements (length of the game) grows; it seems that here also works the famous human short-term memory principle 7 ± 2 [Miller 1956]. The best strategy for human players is to get access to some established source of random binary numbers - use Javascript's function `Math.floor(2*Math.random()` or `window.crypto.getRandomValues()`, download/lookup a table of random binary numbers from `https://www.random.org/` etc and enter your moves from this source.

This mode of playing expels humans. This is 'the table of random numbers vs computer' and this is a rather good test of quality of randomness presented in this table, i.e. evaluation of the quality of the source of randomness.

The descibed above algorithm for searching longest previously already occurred suffix in the list of moves tests, wheather the game automaton is already in cycle, i.e. repeating moves. If it is, it breaks this cycle with probability growing with the length of the cycle – in every move probability, that one of player's (table of random numbers or computer) does not repeat the previously used values (i.e. the probability that the cycle will be broken) is ≈ 0.5. Lack of long cycles is a good evidence for randomness of the sequence.

The described above computer algorithm was tested against three established sources of randomness: Javascript function `random()` (traditional source of randomness in Javascript, works in all major browsers), `window.crypto.getRandomValues()` – the new, 'stronger' Javascript function, and downloading (at the beginning of test) a list of random numbers from `RANDOM.ORG` (https://www.random.org/), where randomness is based on atmospheric noise.

In series of 10 tests in Firefox, every test with 10000 moves (JavaScript `random()` against computer) the number n of occurrences of cycles of length λ is presented the table below. Tests for other opponents: `window.crypto.getRandomValues()` and RANDOM.ORG produced similar results.

Table. 2. Occurrence of cycles and their length in series of 10000 tests:

λ	2	3	4	5	6	7	8	9	10	11	12
n	982	932	756	416	153	46	11	5	3	1	0

The computer played quite well against all three opponents. Below are results of 10 cumulative tests 'player1 = `window.crypto.getRandomValues()`, player2 = computer', each for 10000 moves executed in Google Chrome. While in the first 5 series computer had difficulties, but after that learned constanly and at the end clearly outperformed the opponent.

Table. 3. Cumulative results of 3x10 tests à 10000 moves, player1 = `window.crypto.getRandomValues()`, player2 = `computer`.

Better player1						Better player2						Draw					
Firefox			Chrome			Firefox			Chrome			Firefox			Chrome		
0	1	1	1	0	0	1	0	0	0	1	1	0	0	0	0	0	0
1	1	1	1	1	0	1	1	1	1	1	2	0	0	0	0	0	0
2	2	2	2	1	1	1	1	1	1	2	2	0	0	0	0	0	0
2	2	2	3	2	1	2	2	2	1	2	3	0	0	0	0	0	0
3	2	2	3	2	1	2	3	3	2	3	4	0	0	0	0	0	0
4	3	3	3	2	1	2	3	3	3	4	5	0	0	0	0	0	0
5	4	4	3	3	1	2	3	3	4	4	6	0	0	0	0	0	0
6	5	5	3	3	1	2	3	3	5	5	7	0	0	0	0	0	0
6	5	5	3	3	1	3	4	4	6	6	8	0	0	0	0	0	0
7	6	6	3	4	2	3	4	4	7	6	8	0	0	0	0	0	0

This and results of other similar tests (computer against Javascript's `random()` or table of random values from RANDOM.ORG – their randomness is created from atmospheric noise) show, that the described above algorithm quite well handles these sources of randomness, i.e. its own randomness is on the same leval. From the above table it is seen, that there are some differences in browsers – in Chrome computer outperformed the conventional sources of randomness, but in Firefox the result was opposite – implementation of the function `window.crypto.getRandomValues()` differs in these browsers.

Creating randomness requires memory. In the above experiments the longest subsequence was searched from the whole list of played states, i.e. the algorithm could have as much memory as it needed. If the available memory is restricted, the results are slightly worse, but the difference is marginal. Below are results of tests, if access to memory was restricted - computer could use only the last half of the list of moves (he 'forgets' the earlier moves).

Table. 4. Cumulative results of 3x10 tests à 10000 moves, player1 = window.crypto.getRandomValues(), player2 = computer; computer could use only the last half of the list of made moves.

Better player1						Better player2						Draw					
Firefox			Chrome			Firefox			Chrome			Firefox			Chrome		
1	1	1	0	1	0	0	0	0	1	0	1	0	0	0	0	0	0
2	1	2	1	2	1	0	1	0	1	0	1	0	0	0	0	0	0
2	1	2	2	2	2	1	2	1	1	1	1	0	0	0	0	0	0
3	2	3	3	2	2	1	2	1	1	2	2	0	0	0	0	0	0
3	3	3	4	2	3	2	2	2	1	3	2	0	0	0	0	0	0
3	3	4	4	3	3	3	3	2	2	2	3	0	0	0	0	0	0
3	4	4	4	3	4	4	3	3	3	4	3	0	0	0	0	0	0

4	4	5	4	4	5	4	4	3	4	4	3	0	0	0	0	0	0
4	5	6	5	5	5	5	4	3	4	4	4	0	0	0	0	0	0
4	5	6	6	6	5	6	5	4	4	4	5	0	0	0	0	0	0

7 M-ARY SEQUENCES

The game 'even-odd' works with binary sequences, but it is easy to design similar static games for investigating randomness of m-ary sequences, i.e. sequences of of m-ary residues 0,1,...,m-1, m > 2. Well-known example of such a game is the game rock-paper-scissors, m = 3, where players inputs are compared in one-directional order.

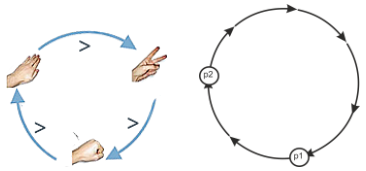

Fig. 2. Decision schema for the rock-paper-scissors game (m = 3) and for the general case (m=7).

In m-ary game players select one of integers 0,1,...,m-1, m > 2; winner is the player, who can 'shoot' the other, if ranges of their wepons are less than m/2 (the shooting is possible only in one direction), i.e. on the above schema for m=7 winner is player p1 (variations of this game are also available in http://staff.ttu.ee/~jaak/games). The same way as above it is easy to show that the optimal strategy (the Nash equilibrium) for such games is also uniform randomness, thus the above algorithm wins against human players, who often use persistent cyclic motions, predicted by the evolutionary game theory because of humans bounded rationality [Wang,Hu,Zhou 2014].

8 CONCLUSIONS

Games are a convenient and easy to use environment for comparing quality of sources of randomness and generating 'new' randomness. Simple games can also be used e.g. for access control, filtering bots from human users (the function of Google reCaptcha-s). The randomness generated by gameplay can be used for generating encryption keys for participants of multi-user games or for enterprizes practizing BYOK methods – no need for downloading keys from server. Creation of new randomness and entropy would have many applications in in our over-organized non-random environment, where entropy/randomness is becoming an endangered species.

REFERENCES

BSA. Encryption: Why It Matters. Retrieved May 9, 2018 from http://encryption.bsa.org/

Fortinet 2018. Data Breaches Are A Growing Epidemic. How Do You Ensure You're Not Next? Retrieved May 08,2018 from https://www.fortinet.com/blog/threat-research/data-breaches-are-a-growing-epidemic--how-do-you-ensure-you-re-n.html

Cisco 2018. Encrypted Traffic Analytics White Paper. Retrieved May 9 2018 from https://www.cisco.com/c/dam/en/us/.../nb-09-encrytd-traf-anlytcs-wp-cte-en.pdf

SendItCertified 2018. Growth of Encryption Market. Retrieved June 10 2018 from www.senditcertified.com/growth-of-encryption-market/

Ponemon 2018. Global Encryption Trends study. Retrieved May 13 2018 from go.thalesesecurity.com/rs/.../2018-Ponemon-Global-Encryption-Trends-Study-ar.pdf

Doom 1997. id-Software/DOOM, Retrieved on May 28, 2018 from https://github.com/id-Software/DOOM/blob/master/linuxdoom-

1.10/m_random.c
Kolmogorov, A.N. (1965). Three Approaches to the Quantitative Definition of Information. Problems Inform. Transmission. 1 (1): 1–7.
Martin-Löf, P. (1966). The definition of random sequences. Information and Control. 9 (6): 602–619. doi:10.1016/s0019-9958(66)80018-9
Schnorr, C. P. (1971). A unified approach to the definition of a random sequence. Mathematical Systems Theory. 5 (3): 246–258. doi:10.1007/BF01694181
NIST (2016). Random Bit Generation. Retrieved on June 11 2018 from https://csrc.nist.gov/projects/random-bit-generation
G. Marsaglia, W. W. Tsang (2002). Some Difficult-to-pass Tests of Randomness, Journal of Statistical Software, 7:3
Matching Pennies. Retrieved on May 11 2018 from http://matchingpennies.com/matching_pennies/
Nash 1950. Equilibrium points in n-person games. PNAS January 1, 1950. 36 (1) 48-49, Retrieved on May 11 2018 from http://www.pnas.org/content/36/1/48
Babichenko Rubinstein 2016. Communication complexity of approximate Nash equilibria. arXiv.org > cs > arXiv:1608.06580, Retrieved May 13 2018 from https://arxiv.org/abs/1608.06580
Henno 2017. Information and Interaction.
Information Modelling and Knowledge Bases XXVIII
IOS Press 2017, pp 426-449
Miller 1956. Miller, G. A. The magical number seven, plus or minus two: Some limits on our capacity for processing information. Psychological Review. 63 (2), pp 81–97.
Zhijian Wang, Bin Xu & Hai-Jun Zhou (2014). Social cycling and conditional responses in the Rock-Paper-Scissors game. Nature, Scientific Reports vol. 4. Retrieved May 13 2018 from https://www.nature.com/articles/srep05830